Ecommerce Database Management System

This project was developed under the curriculum of the Database Management Systems (DBMS) The objective is to design and implement a robust and scalable e-commerce database using PostgreSQL. The project simulates a real-world online shopping environment with a specific focus on enabling small-scale sellers to transition from traditional offline commerce to a structured and accessible online platform.

Key Features:

- Strong enforcement of data consistency and integrity
- Efficient query execution and indexing
- Transactional operation support
- Ease of access and usage for customers and sellers

Advanced PostgreSQL Features: The database system is enhanced through the integration of advanced PostgreSQL capabilities:

- Triggers to automate stock adjustments and enforce business rules
- Views to simplify complex queries and restrict sensitive information
- Derived attributes for real-time calculations
- Stored procedures for reusable logic
- Analytical queries for personalized product recommendations

Analytical Extension: The project extends beyond database design to incorporate a statistical and visual analysis layer using Python (Pandas, Seaborn, Matplotlib). This enables deeper insights into customer behavior and sales performance.

Applied Statistical Techniques:

- Pearson Correlation: To evaluate relationships between product price and customer ratings
- Linear Regression: To analyze trends such as customer age vs. order amount
- Chi-Square Goodness-of-Fit: To determine distribution patterns in product ratings

A significant analytical strategy employed was drill-down analysis. For example, initial boxplots showed uniform order behavior across age groups. However, upon zooming in on customers aged 40+, linear regression revealed a strong positive correlation between age and order value — an insight hidden at the overview level.

Project Scope and Functional Description: This database system supports a wide range of e-commerce operations:

- Product browsing and filtering by category
- Account management for customers
- Cart functionality including item management and total computation
- Order tracking and multi-mode payment support
- Review and rating submissions
- Seller-side inventory updates and sales reporting
- Transactional integrity and rollback mechanisms

Functional Requirements:

- Account management for customers
- Product browsing, search, and categorization
- Cart and wishlist operations
- Order placement, tracking, and review system
- Seller dashboard with product and stock management
- Multiple payment options (COD, UPI, Card)
- Isolated views and access control for customers and sellers
- Trigger-based automation of inventory and order amount updates
- Business insights via statistical and visual analysis

Normalization Process: Normalization was performed to eliminate redundancy and improve data consistency. The process followed the standard progression from 1NF to 3NF.

Example: Customer Table Unnormalized form (UNF): Customer_ID | Name | Phone1 | Phone2 | Address

1NF:

- Ensure atomic values (split multivalued attributes like Phone1 and Phone2) Customer_ID | Name | Phone | Address

2NF:

- Remove partial dependencies (if any composite primary key is used)
- In this case, single-column PK: no partial dependency, so 2NF satisfied

3NF:

- Remove transitive dependencies
- If Address includes City, State, Zip → separate into Location table Customer Table: Customer_ID | Name | Phone | Location_ID Location Table: Location_ID | City | State | Zip

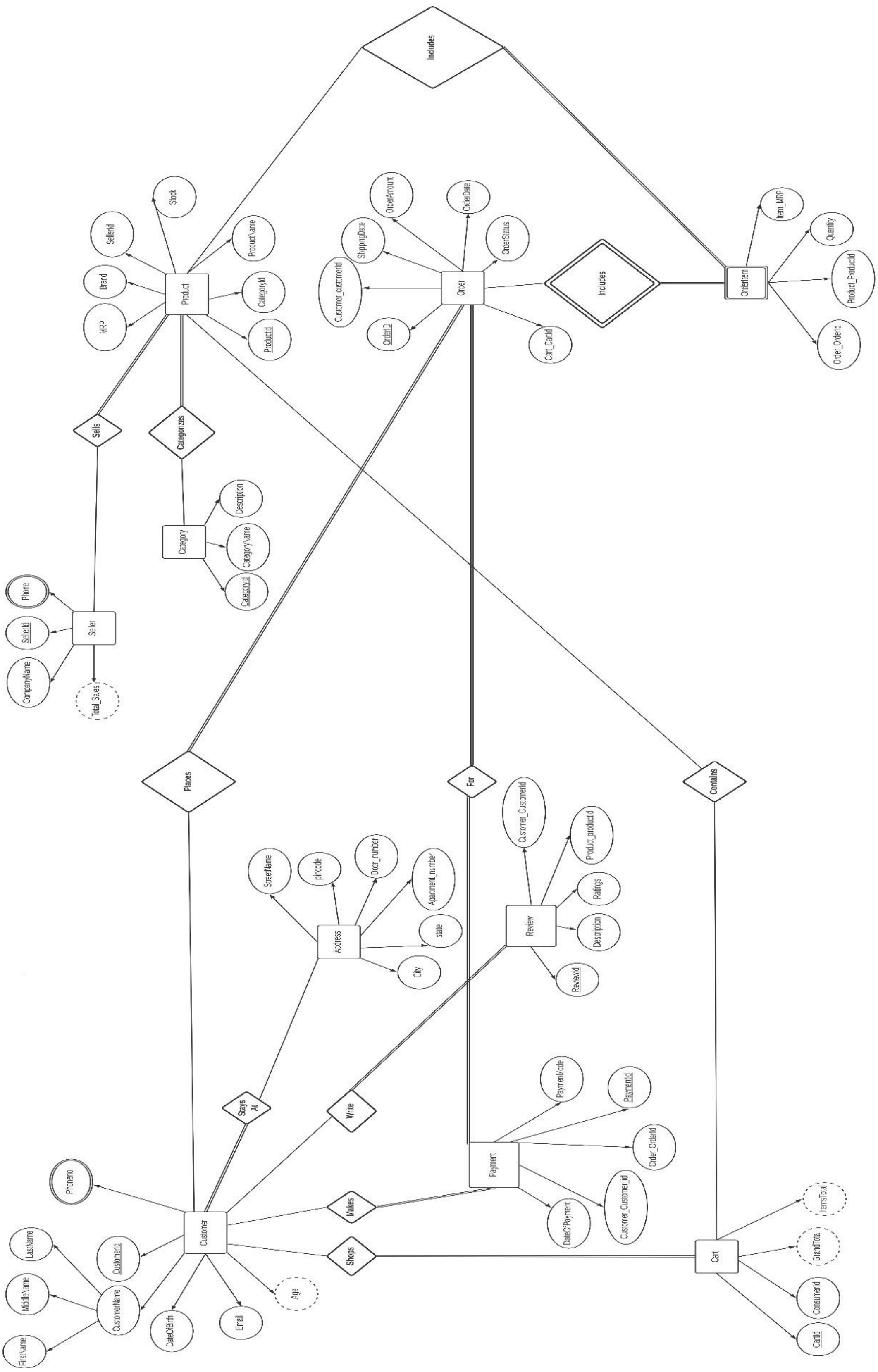
Similar normalization was applied to Orders, Products, and Seller_Product tables to ensure:

- Each table has a primary key
- No multivalued or composite fields
- Every non-key attribute depends only on the key

Future Enhancements: To support real-time analytics and scalable deployment, the following AWS-based enhancements are proposed:

- Kinesis and Firehose for real-time data streaming and ingestion
- AWS Glue and Data Catalog for schema management and ETL
- Amazon S3 for data storage
- Athena for serverless querying
- Lambda for event-driven automation
- Integration of dashboards using Streamlit or Power BI for business reporting

ER-Diagram



Includes
Product
Stock
SellerId
Brand
ProductName
CategoryId
ProductId
MRP
Order
OrderAmount
OrderDate
ShippingDate
OrderStatus
Customer_customerId
OrderId
Cart_CartId
Includes
OrderItem
Item_MRP
Quantity
Product_ProductId
Order_OrderId
Sells
Categorizes
Category
Description
CategoryName
CategoryId
Seller
Phone
SellerId
CompanyName
Places
For
Contains
Review
Customer_CustomerId
Product_productId
Ratings
Description
ReviewId
Address
StreetName
pincode
Door_number
Apartment_number
state
City
Stays At
Write
Payment
PaymentMode
PaymentId
Order_OrderId
Customer_Customer_id
DateOfPayment
Makes
Shops
Customer
Phoneno
CustomerId
CustomerName
LastName
MiddleName
FirstName
DateOfBirth
Email
Age
Cart
ItemsTotal
GrandTotal
ConsumerId
CartId

Normalization Process

To ensure data consistency, eliminate redundancy, and avoid update anomalies, the database design was normalized through the standard three normal forms: 1NF, 2NF, and 3NF.

We demonstrate the process using a simplified example of the Customer table.

Step 1: Unnormalized Form (UNF)
Unnormalized data may contain multivalued or composite attributes.

Example:

Customer_ID	Name	Phone Numbers	Address
101	Krisha S	9876543210, 8765432109	Gandhinagar, GJ

Issues:

- Phone Numbers contains multiple values
- Address is a composite field

◈ Step 2: First Normal Form (1NF)
All attributes must be atomic (indivisible). Multivalued fields are split into separate rows.

Customer_ID	Name	Phone	Address
101	Krisha S	9876543210	Gandhinagar, GJ
101	Krisha S	8765432109	Gandhinagar, GJ

Now, each attribute contains a single value. This satisfies 1NF.

◈ Step 3: Second Normal Form (2NF)
2NF requires the table to be in 1NF and that all non-key attributes are fully functionally dependent on the entire primary key.

If the table has a composite primary key (e.g., Order_ID, Product_ID), partial dependencies must be removed.

In this case, Customer_ID is the primary key, and all attributes depend entirely on it — so 2NF is already satisfied.

◈ Step 4: Third Normal Form (3NF)
A table is in 3NF if it is in 2NF and has no transitive dependencies — i.e., non-key attributes should not depend on other non-key attributes.

If Address contains City, State, and Zip, these should be factored out into a separate Location table.

Split:

Customer Table:

Customer_ID	Name	Phone	Location_ID
101	Krisha S	9876543210	L001

Location Table:

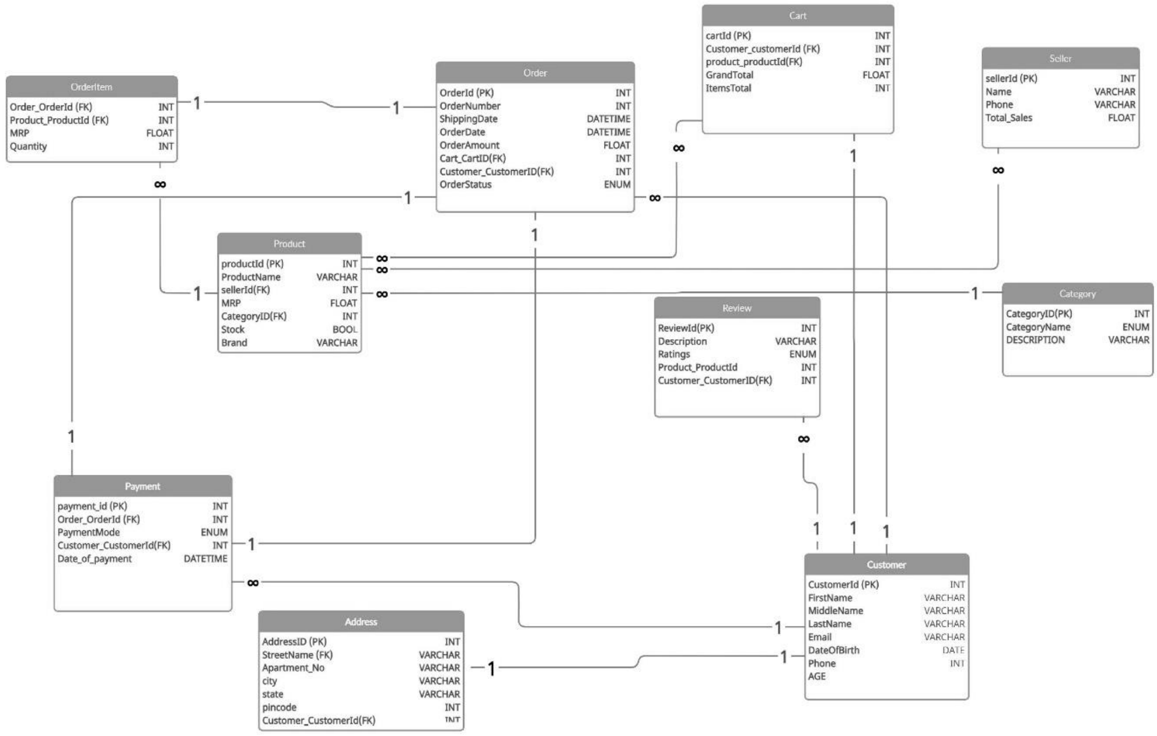
Location_ID	City	State	Zip
L001	Gandhinagar	GJ	382010

This eliminates redundancy and satisfies 3NF.

 Similar normalization steps were applied to:

- Product: Removed embedded Category → separate Category table
- Orders: Split delivery address → reference Location table
- Seller_Product: Many-to-many → bridge table with stock and price info

Relational Database Schema



The database schema includes entities and their relationships to support e-commerce operations. Below is the schema with entities, attributes, and their types.

Entities and Attributes

ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
Customer	Customer_CustomerId Name Email DateOfBirth Phone Age	Simple Composite Simple Simple Multivalued Derived	Strong
Order	OrderId ShippingDate OrderDate OrderAmount Cart_CartID	Simple Simple Simple Simple Simple	Strong
OrderItem	Order_OrderId (PK) Product_ProductId(FK) MRP Quantity	Simple Simple Simple Simple	Weak
Product	productId (PK) ProductName sellerId MRP CategoryID Stock Brand	Simple Simple Simple Simple Simple Simple Simple	Strong
Review	ReviewId(PK) Description Ratings Product_ProductId Customer_CustomerID	Simple Simple Simple Simple	Strong
Cart	cartId (PK) Customer_customerId(FK)	Simple Simple	Strong

ENTITIES	ATTRIBUTES	ATTRIBUTE TYPE	Entity Type
	GrandTotal ItemsTotal	Derived Derived	
Category	CategoryID(PK) CategoryName DESCRIPTION	Simple Simple Simple	Strong
seller	sellerId (PK) Name Phone Total_Sales	Simple Simple Multivalued Derived	Strong
Payment	payment_id Order_OrderId PaymentMode Customer_CustomerId PaymentDate	Simple Simple Simple Simple Simple	Strong

Entity Relationships and Cardinality

ENTITIES	RELATION	CARDINALITY	TYPE OF PARTICIPATION
Customer Address	Stays At	OneToOne	Total Partial
Customer Cart	Shops	OneToOne	Partial Total
Customer Order	Places	OneToMany	Partial Total
Customer Payment	Makes	OneToMany	Partial Total
Customer Review	Write	OneToMany	Partial Total
Seller Product	Sells	ManyToMany	Partial Total
Category Product	Categorizes	OneToMany	Partial Total
Cart Product	Contains	ManyToMany	Partial Partial
Product OrderItem	Includes	OneToMany	Partial Total
Order OrderItem	Includes	OneToOne	Partial Total
Payment Order	For	OneToOne	Total Total

DDL Queries

Below are the Data Definition Language (DDL) queries to create the database and tables using PostgreSQL syntax.

```sql
-- Create Database
CREATE DATABASE ecommerce_db;
\c ecommerce_db;

-- Create Customer Table
CREATE TABLE customer (
    customer_id INTEGER PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    email VARCHAR(100) UNIQUE,
    date_of_birth DATE,
    phone VARCHAR(15),
    age INTEGER GENERATED ALWAYS AS (DATE_PART('year', AGE(date_of_birth))) STORED
);

-- Create Address Table
CREATE TABLE address (
    address_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    street VARCHAR(100),
    city VARCHAR(50),
    pincode VARCHAR(10),
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

```sql
-- Create Cart Table
CREATE TABLE cart (
    cart_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

-- Create Category Table
CREATE TABLE category (
    category_id INTEGER PRIMARY KEY,
    name VARCHAR(50)
);

-- Create Product Table
CREATE TABLE product (
    product_id INTEGER PRIMARY KEY,
    name VARCHAR(100),
    price NUMERIC(10,2),
    stock INTEGER,
    category_id INTEGER,
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- Create Seller Table
CREATE TABLE seller (
    seller_id INTEGER PRIMARY KEY,
    name VARCHAR(100)
```

```sql
);

-- Create Order Table
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    order_date DATE,
    shipping_date DATE,
    order_amount NUMERIC(10,2),
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);

-- Create Payment Table
CREATE TABLE payment (
    payment_id INTEGER PRIMARY KEY,
    order_id INTEGER,
    payment_mode VARCHAR(20) CHECK (payment_mode IN ('COD', 'CreditCard', 'DebitCard',
'UPI')),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

-- Create Review Table
CREATE TABLE review (
    review_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    product_id INTEGER,
    rating INTEGER CHECK (rating BETWEEN 1 AND 5),
    comment TEXT,
```

```sql
    FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create OrderItem Table
CREATE TABLE order_item (
    order_item_id INTEGER PRIMARY KEY,
    order_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create Seller_Product (Many-to-Many relationship between Seller and Product)
CREATE TABLE seller_product (
    seller_id INTEGER,
    product_id INTEGER,
    PRIMARY KEY (seller_id, product_id),
    FOREIGN KEY (seller_id) REFERENCES seller(seller_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);

-- Create Cart_Product (Many-to-Many relationship between Cart and Product)
CREATE TABLE cart_product (
    cart_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
```

```
    PRIMARY KEY (cart_id, product_id),
    FOREIGN KEY (cart_id) REFERENCES cart(cart_id),
    FOREIGN KEY (product_id) REFERENCES product(product_id)
);
```

DML Queries

Insert Queries

Below are sample INSERT queries to populate the database with initial data.

```
-- Insert Customers
INSERT INTO customer (customer_id, first_name, last_name, email, date_of_birth, phone)
VALUES
  (1, 'John', 'Doe', 'john.doe@email.com', '1990-05-15', '1234567890'),
  (2, 'Jane', 'Smith', 'jane.smith@email.com', '1985-08-22', '0987654321');

-- Insert Addresses
INSERT INTO address (address_id, customer_id, street, city, pincode)
VALUES
  (1, 1, '123 Main St', 'New York', '10001'),
  (2, 2, '456 Oak Ave', 'Los Angeles', '90001');

-- Insert Categories
INSERT INTO category (category_id, name)
VALUES
  (1, 'Electronics'),
  (2, 'Clothing');

-- Insert Products
INSERT INTO product (product_id, name, price, stock, category_id)
VALUES
  (1, 'Smartphone', 699.99, 50, 1),
  (2, 'T-Shirt', 19.99, 100, 2);

-- Insert Sellers
```

```
INSERT INTO seller (seller_id, name);

VALUES

   (1, 'TechTrend Innovations'),

   (2, 'FashionHub');


-- Insert Seller_Product Relationships

INSERT INTO seller_product (seller_id, product_id)

VALUES

   (1, 1),

   (2, 2);


-- Insert Carts

INSERT INTO cart (cart_id, customer_id)

VALUES

   (1, 1),

   (2, 2);


-- Insert Cart_Product Relationships

INSERT INTO cart_product (cart_id, product_id, quantity)

VALUES

   (1, 1, 2),

   (2, 2, 3);


-- Insert Orders

INSERT INTO orders (order_id, customer_id, order_date, shipping_date, order_amount)

VALUES

   (1, 1, '2025-06-01', '2025-06-05', 1399.98),

   (2, 2, '2025-06-02', '2025-06-06', 59.97);
```

```sql
-- Insert OrderItems
INSERT INTO order_item (order_item_id, order_id, product_id, quantity)
VALUES
    (1, 1, 1, 2),
    (2, 2, 2, 3);

-- Insert Payments
INSERT INTO payment (payment_id, order_id, payment_mode)
VALUES
    (1, 1, 'CreditCard'),
    (2, 2, 'COD');

-- Insert Reviews
INSERT INTO review (review_id, customer_id, product_id, rating, comment)
VALUES
    (1, 1, 1, 5, 'Excellent smartphone!'),
    (2, 2, 2, 4, 'Comfortable T-shirt.');
```

Sample Queries

Below are example DML queries to demonstrate key operations based on the provided requirements, written in PostgreSQL.

1. **<u>Find products with the highest ratings for a given category (Category: Electronics)</u>**

```
SELECT p.product_id, p.name, AVG(r.rating)::NUMERIC(3,2) AS avg_rating
FROM product p
JOIN review r ON p.product_id = r.product_id
WHERE p.category_id = 1
GROUP BY p.product_id, p.name
ORDER BY avg_rating DESC
LIMIT 5;
```

----

2. **<u>Filter products by brand and price (Price < 100)</u>**

```
SELECT p.product_id, p.name, p.price, s.name AS seller_name
FROM product p
JOIN seller_product sp ON p.product_id = sp.product_id
JOIN seller s ON sp.seller_id = s.seller_id
WHERE p.price < 100
ORDER BY p.price;
```

----

3. **<u>Calculate total price in a customer's cart</u>**

```
SELECT c.cart_id, SUM(p.price * cp.quantity) AS total_amount
FROM cart c
JOIN cart_product cp ON c.cart_id = cp.cart_id
JOIN product p ON cp.product_id = p.product_id
```

WHERE c.customer_id = 1

GROUP BY c.cart_id;

---

4. **Find the best seller for a particular product** (ProductID: 1)

SELECT s.seller_id, s.name, COUNT(oi.order_id) AS total_orders

FROM seller s

JOIN seller_product sp ON s.seller_id = sp.seller_id

JOIN order_item oi ON sp.product_id = oi.product_id

WHERE sp.product_id = 1

GROUP BY s.seller_id, s.name

ORDER BY total_orders DESC

LIMIT 1;

---

5. **List orders to be delivered at a particular pincode**

SELECT o.order_id, o.order_date, o.order_amount

FROM orders o

JOIN customer c ON o.customer_id = c.customer_id

JOIN address a ON c.customer_id = a.customer_id

WHERE a.pincode = '10001';

---

6. **List products with the highest sales on a particular day** (Date: 2025-06-01)

SELECT p.product_id, p.name, SUM(oi.quantity) AS total_sold

FROM product p

JOIN order_item oi ON p.product_id = oi.product_id

JOIN orders o ON oi.order_id = o.order_id

```
WHERE o.order_date = '2025-06-01'

GROUP BY p.product_id, p.name

ORDER BY total_sold DESC

LIMIT 1;
```

----

7. **List categories with the highest sales on a particular day** (Date: 2025-06-01)

```
SELECT c.category_id, c.name, SUM(oi.quantity) AS total_sold

FROM category c

JOIN product p ON c.category_id = p.category_id

JOIN order_item oi ON p.product_id = oi.product_id

JOIN orders o ON oi.order_id = o.order_id

WHERE o.order_date = '2025-06-01'

GROUP BY c.category_id, c.name

ORDER BY total_sold DESC

LIMIT 1;
```

----

8. **List customers who bought the most from a particular seller**

```
SELECT c.customer_id, c.first_name, c.last_name, COUNT(o.order_id) AS total_orders

FROM customer c

JOIN orders o ON c.customer_id = o.customer_id

JOIN order_item oi ON o.order_id = oi.order_id

JOIN seller_product sp ON oi.product_id = sp.product_id

WHERE sp.seller_id = 1

GROUP BY c.customer_id, c.first_name, c.last_name

ORDER BY total_orders DESC

LIMIT 5;
```

---

9. **List orders with non-COD payment modes that are yet to be delivered**

```
SELECT o.order_id, o.order_date, p.payment_mode
FROM orders o
JOIN payment p ON o.order_id = p.order_id
WHERE p.payment_mode != 'COD' AND o.shipping_date > CURRENT_DATE;
```

---

10. **List orders with total amount greater than 5000**

```
SELECT o.order_id, o.order_date, o.order_amount
FROM orders o
WHERE o.order_amount > 5000;
```

---

11. Product Recommendation Queries

```
SELECT oi2.product_id, p.name, COUNT(*) AS purchase_count
FROM order_item oi1
JOIN order_item oi2 ON oi1.order_id = oi2.order_id AND oi1.product_id != oi2.product_id
JOIN product p ON p.product_id = oi2.product_id
WHERE oi1.product_id = 1
GROUP BY oi2.product_id, p.name
ORDER BY purchase_count DESC
LIMIT 5;
```

---

12. **Popular Products in a Category (Based on Reviews)**

```
SELECT p.product_id, p.name, AVG(r.rating) AS avg_rating, COUNT(r.rating) AS total_reviews
FROM product p
JOIN review r ON r.product_id = p.product_id
WHERE p.category_id = 1 -- replace with desired category
```

```sql
GROUP BY p.product_id, p.name

HAVING COUNT(r.rating) >= 2

ORDER BY avg_rating DESC

LIMIT 5;
```

---

13. **Time-Based Sales Analytics: Best-Selling Products Last Month**

```sql
SELECT p.product_id, p.name, SUM(oi.quantity) AS units_sold

FROM product p

JOIN order_item oi ON p.product_id = oi.product_id

JOIN orders o ON oi.order_id = o.order_id

WHERE o.order_date >= date_trunc('month', CURRENT_DATE) - INTERVAL '1 month'

  AND o.order_date < date_trunc('month', CURRENT_DATE)

GROUP BY p.product_id, p.name

ORDER BY units_sold DESC

LIMIT 5;
```

---

14. **Top Customers by Spending**

```sql
SELECT c.customer_id, c.first_name, c.last_name, SUM(o.order_amount) AS total_spent

FROM customer c

JOIN orders o ON c.customer_id = o.customer_id

GROUP BY c.customer_id, c.first_name, c.last_name

ORDER BY total_spent DESC

LIMIT 5;
```

---

15. **Stored Procedure: Top Sellers by Category**

```
CREATE OR REPLACE PROCEDURE top_sellers_by_category(cat_id INTEGER)
LANGUAGE plpgsql
AS $$
BEGIN
  SELECT s.seller_id, s.name, COUNT(oi.order_id) AS total_orders
  FROM seller s
  JOIN seller_product sp ON s.seller_id = sp.seller_id
  JOIN product p ON p.product_id = sp.product_id
  JOIN order_item oi ON oi.product_id = p.product_id
  WHERE p.category_id = cat_id
  GROUP BY s.seller_id, s.name
  ORDER BY total_orders DESC
  LIMIT 5;
END;
```

----

16. **Detect and List Inactive Customers**

```
SELECT c.customer_id, c.first_name, c.last_name
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id
  AND o.order_date >= CURRENT_DATE - INTERVAL '60 days'
WHERE o.order_id IS NULL;
```

----

17. **Frequent Co-Purchased Category Pairings**

```
SELECT c1.name AS category1, c2.name AS category2, COUNT(*) AS co_purchases
FROM order_item oi1
```

```
JOIN product p1 ON oi1.product_id = p1.product_id
JOIN category c1 ON p1.category_id = c1.category_id

JOIN order_item oi2 ON oi1.order_id = oi2.order_id AND oi1.product_id != oi2.product_id
JOIN product p2 ON oi2.product_id = p2.product_id
JOIN category c2 ON p2.category_id = c2.category_id

GROUP BY c1.name, c2.name
ORDER BY co_purchases DESC
LIMIT 10;
```

Trigger: Update Stock After Payment

```
CREATE OR REPLACE FUNCTION update_stock_after_payment()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE product
    SET stock = stock - oi.quantity
    FROM order_item oi
    WHERE oi.product_id = product.product_id
    AND oi.order_id = NEW.order_id;
    RETURN NEW;
END;

CREATE TRIGGER update_stock_trigger
AFTER INSERT ON payment
FOR EACH ROW
EXECUTE FUNCTION update_stock_after_payment();
```

Trigger: Update Order Amount

```
CREATE OR REPLACE FUNCTION update_order_amount()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE orders
    SET order_amount = (
        SELECT SUM(p.price * oi.quantity)
        FROM order_item oi
        JOIN product p ON oi.product_id = p.product_id
        WHERE oi.order_id = NEW.order_id
```

```
  )
  WHERE order_id = NEW.order_id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_order_amount_trigger
AFTER INSERT ON order_item
FOR EACH ROW
EXECUTE FUNCTION update_order_amount();
```

Trigger: Auto-Insert Positive Feedback if Rating ≥ 4

```
CREATE OR REPLACE FUNCTION add_auto_comment()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.comment IS NULL AND NEW.rating >= 4 THEN
    NEW.comment := 'Thanks for the great rating!';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER auto_feedback_trigger
BEFORE INSERT ON review
FOR EACH ROW
EXECUTE FUNCTION add_auto_comment();
```

Statistical Tests and Mathematical Foundations

In this project, We have used statistical techniques to uncover insights from structured e-commerce data. These tests were chosen based on the nature of variables (categorical or continuous), data distribution, and the business questions at hand.

---

1. Pearson Correlation Coefficient (r)

Purpose:
To measure the **linear relationship** between two continuous variables.

Mathematical Formula:

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}$$

Where:

- x_i, y_i: individual sample points
- $\bar{x}, \bar{y}$: means of x and y

Range: $-1 \leq r \leq 1$

Used For:

- Product price vs. rating
- Product price vs. quantity sold

Business Insight:
Shows whether expensive products get better reviews or sell more.

Simple Linear Regression

Purpose:
To model and predict the relationship between an **independent variable (X)** and a **dependent variable (Y)**.

Mathematical Model:

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

Where:

- Y: dependent variable (e.g., order amount)
- X: independent variable (e.g., customer age)
- β_0: intercept
- β_1: slope (change in Y per unit change in X)
- ε: error term

Used For:

- Customer age vs. order amount

Business Insight:
Helps identify which customer age groups are more profitable.

Chi-Square Goodness of Fit Test (χ^2)

Purpose:
To test whether a categorical variable follows a given distribution.

Mathematical Formula:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Where:

- O_i: Observed frequency
- E_i: Expected frequency

Degrees of Freedom: $df = k - 1$, where k = number of categories

Used For:

- Checking if ratings are uniformly distributed

Business Insight:
Detects rating biases or artificial inflation of 5-star reviews.

Boxplots (Visual Test)

Purpose:
A graphical method to display the distribution of data using **five-number summary**:

- Minimum, Q1 (25%), Median (Q2), Q3 (75%), Maximum

Interquartile Range (IQR):

$$IQR = Q_3 - Q_1$$

Outlier Detection:

$$\text{Lower Bound} = Q_1 - 1.5 \times IQR$$
$$\text{Upper Bound} = Q_3 + 1.5 \times IQR$$

Used For:

- Order amount by region, age group, payment mode
- Ratings by category

Business Insight:
Helps visualize distribution, detect outliers, and compare variability.

Data Visualization & Insights

To supplement the database operations and statistical analysis, data visualizations were created using Python libraries including Pandas, Seaborn, and Matplotlib. These visualizations helped uncover trends, anomalies, and business insights that are not immediately evident from raw data alone.

The visualizations served three main purposes:

- To explore relationships between key variables such as age, order value, product price, and rating
- To summarize aggregate patterns such as regional sales, payment preferences, and product popularity
- To support statistical conclusions with visual evidence

Tools Used:

- Python (Jupyter Notebook)

- Seaborn and Matplotlib for static visualizations
- Optional: Tableau for interactive dashboards

🔷 Types of Visualizations Created:

- Bar Charts: for identifying top-selling products and most active regions
- Line Charts: for analyzing monthly sales and order trends over time
- Boxplots: to compare order amounts across customer age groups and payment modes
- Heatmaps: to show correlation between numeric variables (e.g., price vs. rating, age vs. order value)

In our exploratory data analysis, we initially visualized overall customer spending behavior using a boxplot segmented by age groups. At a high level, it appeared that all age groups spent similarly.

However, when we zoomed in and separated customers over age 40 into finer brackets (e.g., 40–49, 50–59, 60+), we discovered a clear upward trend — the older the user, the higher the average order value.

This deeper insight was only visible after drilling down and segmenting the data more precisely. We supported this finding with a linear regression model that showed a statistically significant positive slope between customer age and order amount ($p < 0.05$).